ACTOR RECOGNIZER

Project Activity Mobile Systems M – 2022/2023

Ambrogini Thomas

Table of Contents

1. Abstract 3
2. Cloud Computing and Edge Computing 3
1.1 Cloud Computing 3
1.2 Edge Computing 4
1.3 Cloud Computing vs Edge Computing 5
3. AWS 6
2.1 S3 7
2.2 Lambda Functions 8
2.3 AWS SageMaker 9
2.3.1 Using TensorFlow with SageMaker 10
2.4 AWS Edge 12
2.5 AWS GreenGrass 12
2.5.1 Core Concepts 13
2.5.2 Machine Learning Inference 14
2.5.3 Machine Learning Components 15
4. Face Recognition 15
3.1 Face Detection 16
3.2 Feature Extraction 17
3.3 Recognition 18
5. Implementation 20
4.1 Dataset 20
4.2 Neural Network 21
4.2.1 Training 21
4.3 Deployment on the edge 24
4.3 Results 25

1. Abstract

The objective of this project is to create an actor recognition system using AWS, specifically designed to identify actors within a movie scene. Employing facial recognition techniques, the system will categorize numerous actors based on IMDb's influential actor rankings. The system's key feature lies in its deployment at the edge, enabling efficient processing directly where data is generated.

2. Cloud Computing and Edge Computing

1.1 Cloud Computing

Cloud computing is a general term for anything that involves delivering hosted services over the internet. These services include tools and applications like data storage, servers, databases, networking, and software.
Rather than keeping files on a hard drive or local storage device, cloud-based storage makes it possible to save them to a remote database. As long as an electronic device has access to the web, it has access to the data and the software programs to run it.
Cloud computing is a popular option for people and businesses for a number of reasons including: cost savings, increased productivity, speed and efficiency, performance, and security.
One of the main advantages of cloud computing is that it eliminates the need for substantial initial investments in hardware or the complexities of infrastructure management. Instead, it facilitates the seamless configuration of computing resources, supporting ongoing IT operations. This model ensures immediate resource availability and charges users only for their specific usage.

Figure 1: Cloud computing

1.2 Edge Computing

Edge computing is a decentralized approach to processing data closer to where it's generated, rather than relying on a central location like a cloud server. It involves placing computing resources (such as servers or data centers) closer to the data source, reducing the distance data needs to travel and minimizing latency.
This paradigm allows for faster data processing, real-time analysis, and reduced bandwidth usage by handling data at or near the source. It's particularly useful in situations where immediate processing and response are critical, like in IoT devices, autonomous vehicles, or industries requiring real-time data analysis. Edge computing complements cloud computing by offloading some tasks from centralized servers, enhancing efficiency, and enabling faster decision-making at the network's edge.

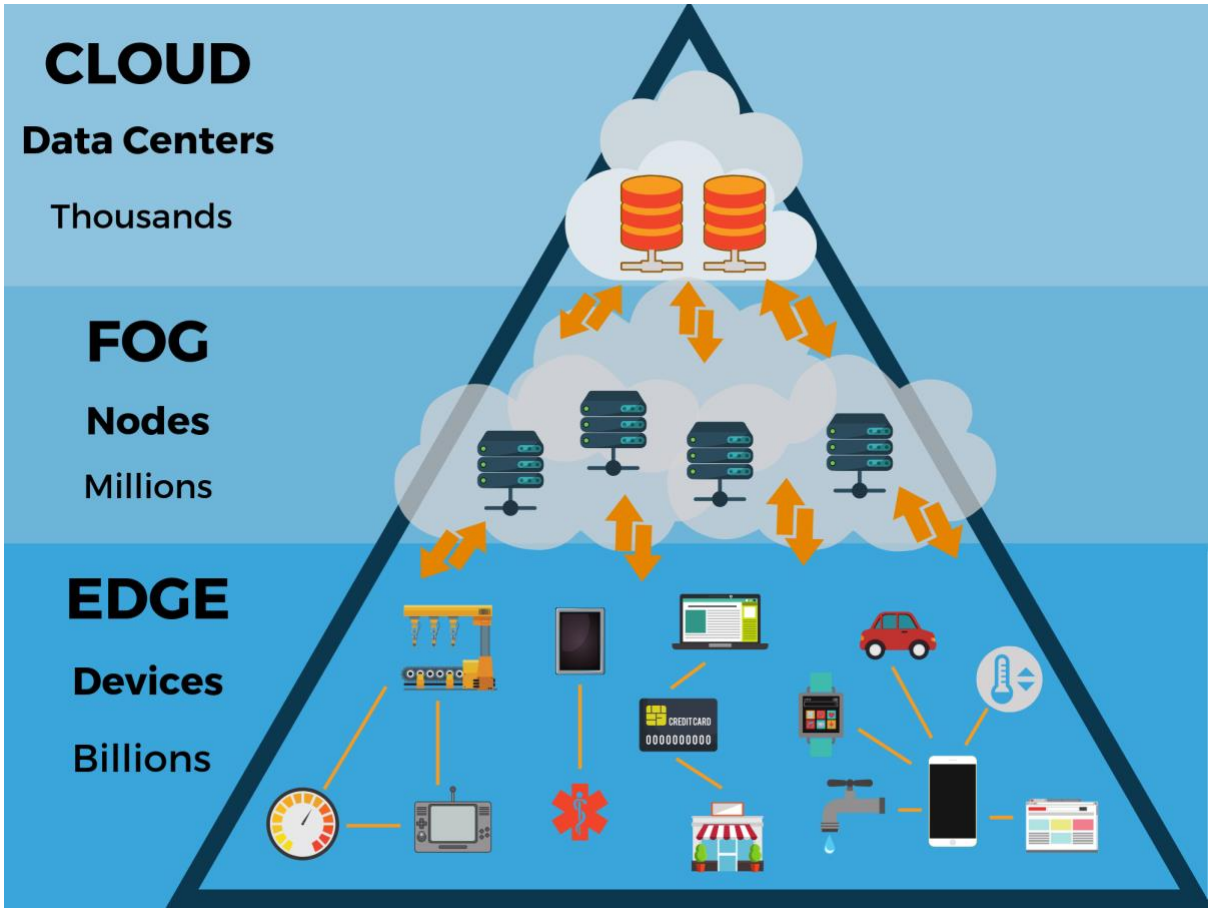

Figure 2: Edge Computing

1.3 Cloud Computing vs Edge Computing

The rise of edge computing doesn't signify the replacement of cloud computing. Instead, in the aspect of network infrastructure, business operations, application deployment, and intelligence augmentation, these two paradigms should work together. While edge nodes process data at its source, the cloud is still needed to do analysis and obtain more meaningful analysis results.

When IoT devices generate large amounts of data, sending it all to the cloud can overwhelm the system. Introducing edge computing into this scenario can relieve some of the pressure of the by handling some data processing tasks closer to where the data is generated. This integration allows for more efficient management of the workload, balancing the distribution of data processing between edge devices and the centralized cloud infrastructure.

	Applicable situation	Network bandwidth pressure	Real-time	Calculation mode
Cloud computing	Global	More	High	Large scale centralized processing
Edge computing	Local	Less	Low	Small scale intelligent analysis

Figure 3: Differences between edge and cloud

Cloud computing is characterized by its ability to handle vast volumes of data, doing in-depth analysis and non-real-time data processing, like supporting business decision-making and various other fields.
Edge computing focuses on the local, and can play a better role in small-scale, real-time intelligent analysis, such as meeting the real-time needs of local businesses.

3. AWS

Amazon Web Services (AWS) is a cloud service provider offering a vast array of on-demand computing resources and services. It provides businesses, organizations, and individuals with access to scalable and flexible computing power, storage solutions, and a wide range of cloud-based applications. AWS offers different functionalities such as computing power through services like Amazon EC2, storage with Amazon S3, databases via Amazon RDS, and machine learning with Amazon SageMaker, AWS enables users to deploy applications quickly and efficiently while paying only for the resources used.

Figure 4: AWS logo

In the following chapters I am going to introduce the main components of AWS used in this project.

2.1 S3

Amazon Simple Storage Service (S3) is a highly scalable and secure cloud storage service offered by Amazon Web Services (AWS). It provides a platform for storing and retrieving vast amounts of data, offering durability, availability, and performance at a low cost.
S3 operates as an object storage system, allowing users to store diverse data types, including files, images, videos, and backups, organized into containers called "buckets."
It enables data management and access via a simple web interface or programmatically through APIs.

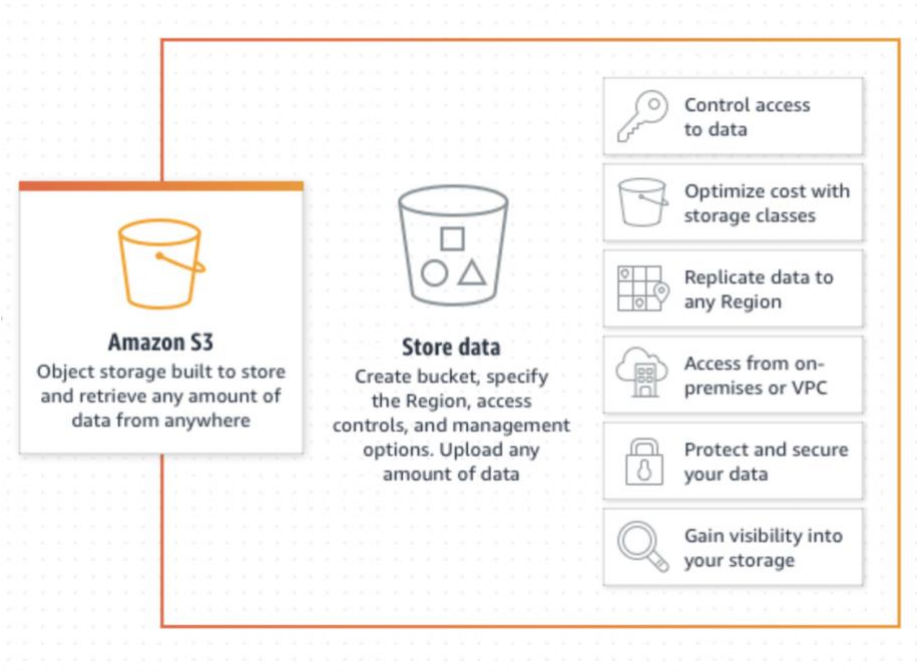

Figure 5: AWS S3 primary functionalities

The S3 buckets serves as a fundamental component in every kind of Artificial intelligence application, since there is a big demand for data in this domain. It’s not important only for storing the dataset used for the training and testing of the

neural network, but buckets are even used to store models, once they are trained, and all the artifacts connected to them.
Furthermore, it is perfectly integrated with every other AWS service, working almost as a base component for every other service. For instance, AI models trained on big datasets stored in S3 effortlessly interface with various AWS AI services like Amazon Rekognition or Amazon SageMaker.

2.2 Lambda Functions

AWS Lambda is a serverless computing service designed to execute code in response to various events while handling all underlying compute resources automatically. Its key features include: integration with other AWS services for custom logic application, enabling the extension of AWS resources like Amazon S3 and DynamoDB.

Lambda simplifies the process of creating custom backend services triggered by APIs or specific events. With Lambda, developers can bring their own code, including third-party libraries, frameworks, and native languages, allowing flexibility in development. The service automates infrastructure administration, ensuring high availability and fault tolerance across AWS Regions.
Lambda offers fine-grained control over performance, scalability, and cost-effectiveness by scaling based on the incoming request rate and charging based on actual usage, making it an ideal choice for various application needs.

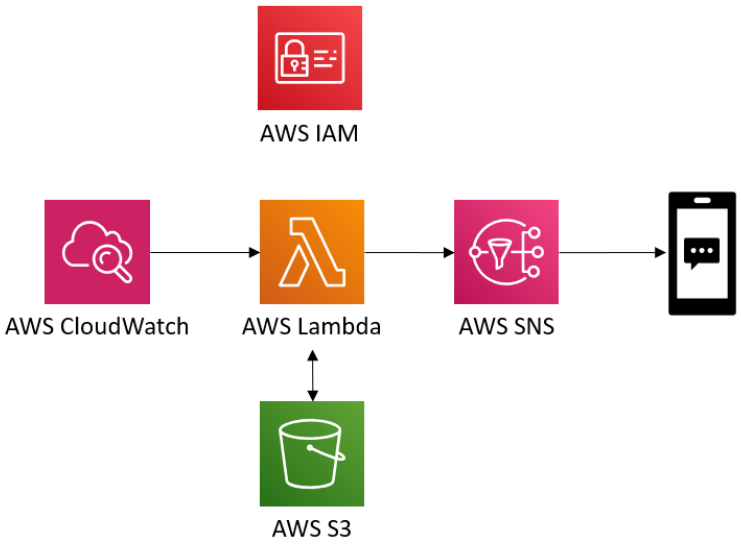

Figure 6: AWS Lambda

2.3 AWS SageMaker

Amazon SageMaker is a machine learning service where data scientists and developers can quickly and easily build and train machine learning models, then deploy them directly into a production-ready hosted environment. It includes standard machine learning methods geared for use in a distributed setting with exceptionally substantial data sets. SageMaker offers versatile distributed training alternatives that adapt to your workflows.

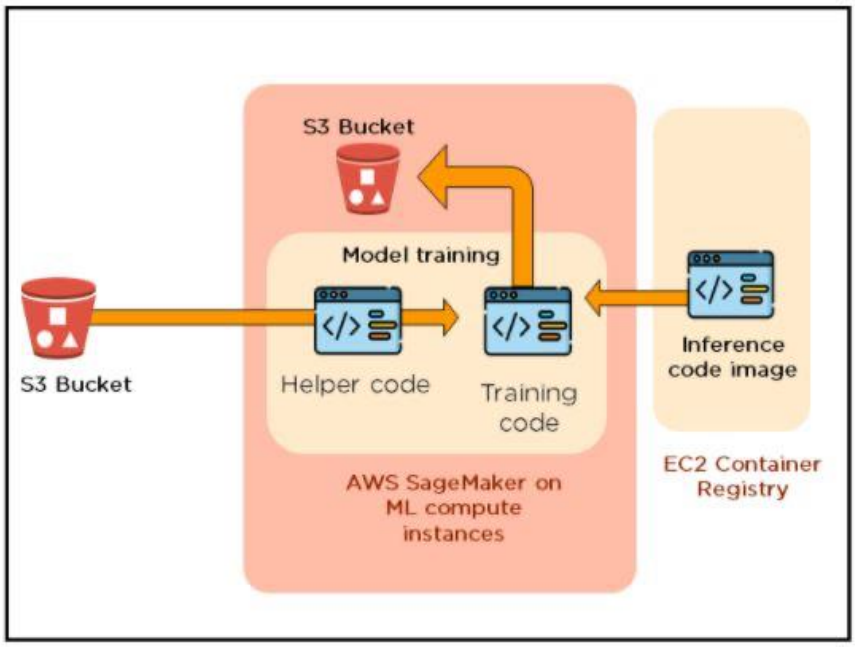

Figure 7: Sagemaker workflow

Amazon SageMaker uses Amazon EC2 to create a fully managed, meaning that it's not a container that you have to manage, machine learning environment. At its core is the versatile Jupyter Notebook web application, empowering developers to collaboratively share live code. Within SageMaker, these notebooks are equipped with essential drivers, packages, and libraries for the most used deep learning frameworks. The main advantage of this is that the user doesn't have to manage anything about the environment and all the python libraries needed because they come bundled with the EC2 container and the jupyter notebook, where you can specify a kernel with the standard libraries already installed.

Moreover, developers possess the flexibility to employ custom-built algorithms, either within supported ML frameworks or encapsulated within Docker container

images. SageMaker accesses data stored in Amazon S3. This gives a lot of modularity and reusability to solutions created from developers, since they just have to be put inside a container and they can be used by other developers.

2.3.1 Using TensorFlow with SageMaker

TensorFlow is an open-source machine learning library developed by Google, known for its flexibility and scalability in creating neural networks and deep learning models. It offers a comprehensive suite of tools and resources for building, training, and deploying machine learning algorithms efficiently and it is one of the most used frameworks in the world.
Users can access TensorFlow's extensive suite of tools and functionalities directly within SageMaker's environment. If the TensorFlow functionalities are used in sagemaker there are some little differences between using the SageMaker version and TensorFlow in a local environment.

```
sagemaker_session = sagemaker.Session()
role = sagemaker.get_execution_role()

estimator = TensorFlow(entry_point='training_script.py',
                       role=role,
                       instance_count=1,
                       instance_type='ml.m4.xlarge',
                       framework_version='2.5.0',
                       py_version='py38',
                       output_path='s3://bucket-name/model-output/',
                       hyperparameters={'epochs': 10, 'batch-size': 64})

estimator.fit({'training': 's3://bucket-name/training-data/'})
```

Figure 8: SageMaker Training code

First of all there is the need of defining the SageMaker session and the role assigned to SageMaker, which defines all the resources accessible from SageMaker (e.g., S3 buckets).
For the actual model a TensorFlow Estimator must be defined where the developer has to specify: an entry point script or training script, the number of instances on which the training will be done and what type of instance must be used. One of the features of SageMaker is the ability to train models on different types of instances, enabling users to select the computing resources that best suit their needs. AWS offers a wide range of instance types, from smaller, cost-

effective instances suitable for initial experimentation to high-performance instances with GPU or distributed computing capabilities for training complex models. By using better instances the training time will be shorter, but the cost will be also higher.
When the estimator start the training process, the data on which the training will be done can be passed through the URI of an S3 bucket containing the dataset. Also the model will be saved on an S3 bucket specified with the output_path field inside the estimator.

Training Script

A training script is a Python script that contains the instructions for training a machine learning model. This script typically includes the definition of the model architecture, data preprocessing steps, training loop, evaluation metrics, and model saving procedures. It encapsulates the entire process of training a model on a dataset.
In Amazon SageMaker, when using the TensorFlow Estimator, the training script is specified as the entry point. This script dictates how the model is trained, what data is used, and how the training process progresses.

```
data_path = '/opt/ml/input/data/train/'
dataset = np.load(data_path + 'dataset.npz')

# Process the input
#
#

# Define the model to be used
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.Dense(number_of_classes, activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])

model.fit(x_train, y_train, epochs=args.epochs, validation_data=(x_test, y_test))

model.save('/opt/ml/model/2')
```

Figure 9: Training Script example

Using this approach all the already existing software developed and tested locally can be used without substantial modifications.

As it's possible to see from code snippet above, the dataset passed through the URI of the S3 bucket in the SageMaker code will be found locally in the container used, in the following path: `/opt/ml/input/data/train`.

2.4 AWS Edge

AWS extends its powerful cloud computing capabilities to the edge with a framework designed to revolutionize data processing and service delivery. AWS edge services deliver data processing, analysis, and storage close to the endpoints, allowing to deploy APIs and tools to locations outside AWS data centers.
One of the main advantages of AWS edge is the possibility of integration and reusability of AWS resources. This integration allows businesses to easily deploy, on edge devices, infrastructures, services and tools, which they have already built in the AWS ecosystem.
With the ability to reuse established components such as APIs, tools, and configurations, organizations can achieve faster time-to-market for edge computing applications. Additionally, the unified management and familiar interfaces across both edge and cloud environments simplify operations, enhancing efficiency and reducing the learning curve for deploying edge solutions within the AWS framework.

2.5 AWS GreenGrass

AWS IoT Greengrass is an open source Internet of Things (IoT) edge runtime and cloud service that helps you build, deploy and manage IoT applications on your devices. You can use AWS IoT Greengrass to build software that enables your devices to act locally on the data that they generate and filter and aggregate device data.
AWS Greengrass facilitates the deployment of AWS Lambda functions, machine learning inference, and other cloud services directly onto edge devices, enabling them to operate independently or together with the cloud.
Greengrass devices can also communicate securely with AWS IoT Core and export IoT data to the AWS Cloud.

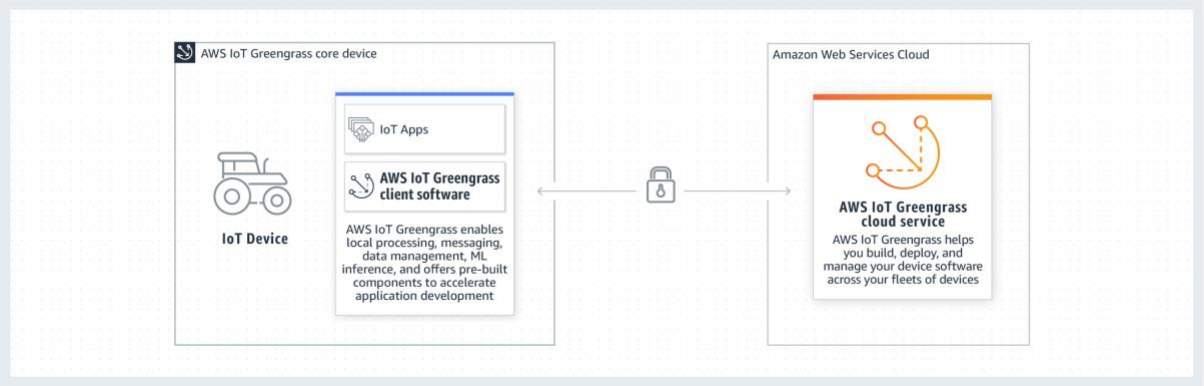

Figure 10: AWS GreenGrass

2.5.1 Core Concepts

GreenGrass Core Device

A device that operates using the AWS IoT Greengrass Core Software, a component that runs on edge devices, such as a Raspberry Pi, and allows local execution of AWS Lambda functions and AWS IoT Device Shadow, among other services.

Thing Groups

Thing Groups in AWS IoT Greengrass provide a structured way to organize and manage multiple devices or Greengrass core devices within the AWS IoT ecosystem. These groups allow to manage devices based on shared characteristics, functionalities or deployment location.

All the elements of a group shares access control policies to interact with other AWS resources, making easier the administration of security.

GreenGrass Client Device

A device that connects to and communicates with a Greengrass core device over MQTT. A Greengrass client device is an AWS IoT thing. The core device can process, filter, and aggregate data from client devices that connect to it. The core device can be configured to relay MQTT messages between client devices, the AWS IoT Core cloud service, and Greengrass components.

GreenGrass Component

A component is a software module that runs on AWS IoT Greengrass core devices. Components enable you to create and manage complex applications as discrete building blocks that you can reuse from one Greengrass core device to another. Every component is composed of a *recipe* and *artifacts*.

- **Recipes**
  The recipe file defines metadata, configuration parameters, dependencies, and the component's lifecycle, specifying installation, execution, and shutdown commands. Recipes can be structured in JSON or YAML formats.

- **Artifacts**
  Artifacts represent component binaries, encompassing scripts, compiled code, and various files consumed by the component, including those from dependencies.

2.5.2 Machine Learning Inference

AWS IoT Greengrass simplifies the execution of machine learning inference directly on devices, using models developed, trained, and fine-tuned in the cloud. It provides the versatility to employ machine learning models trained within Amazon SageMaker or to utilize existing pretrained models stored in Amazon S3.
AWS IoT Greengrass enhances the deployment of machine learning at the edge by enabling real-time processing, reducing latency, and ensuring privacy and security by keeping sensitive data local. It also supports the continuous evolution of models with over-the-air updates, ensuring devices stay updated with the latest advancements without interrupting operations.

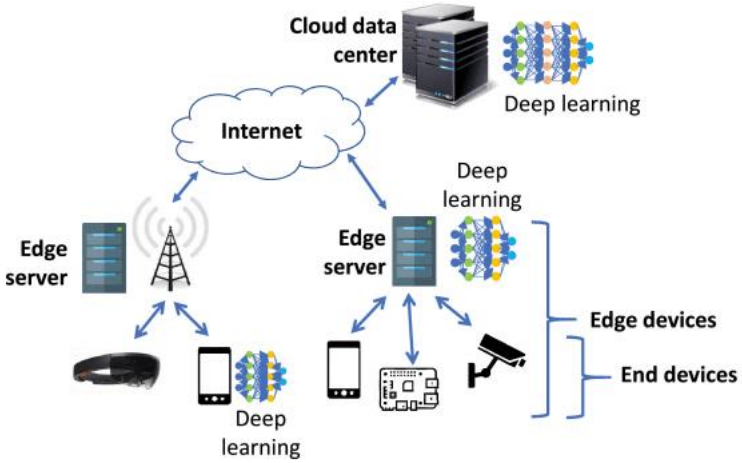

Figure 11: machine learning in the edge

2.5.3 Machine Learning Components

In AWS IoT Greengrass, you can configure machine learning components to customize how you perform machine learning inference on your devices with the inference, model and runtime components as the building blocks.

4. Face Recognition

Face recognition, a cutting-edge technology in the domain of computer vision and artificial intelligence, operates through a series of stages aimed at identifying or verifying individuals based on their facial characteristics.
The process of face recognition follows a specific workflow:

- Face detection.
- Feature extraction.
- Recognition.

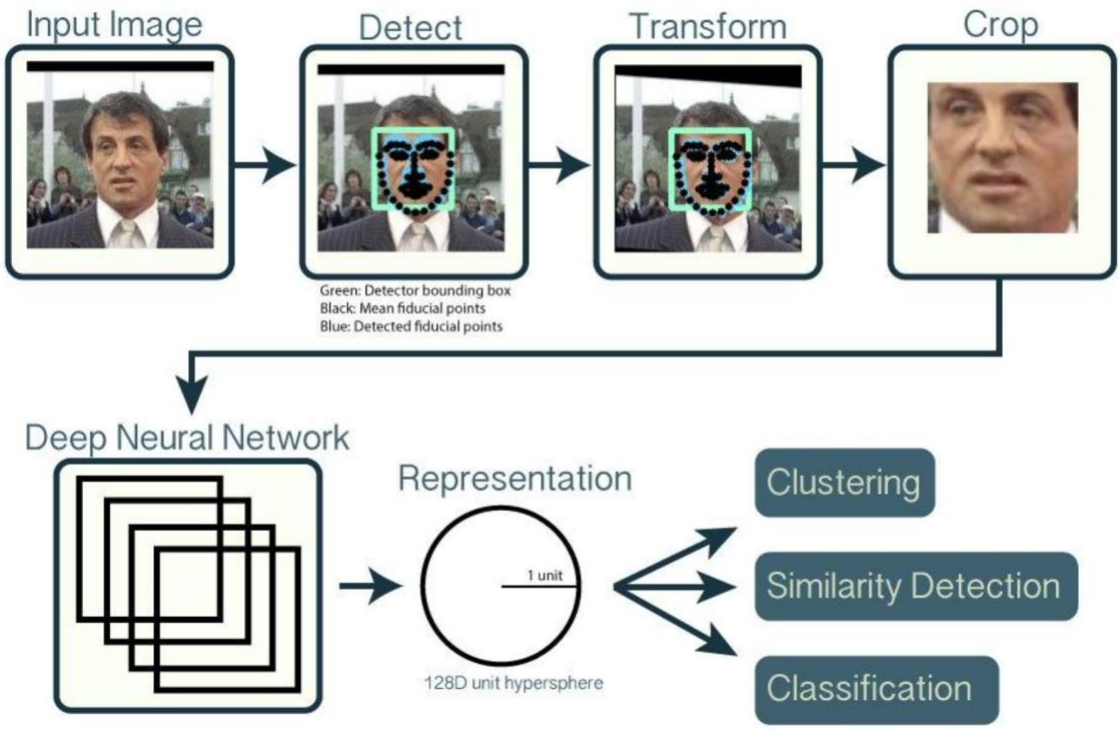

Figure 12: Face Recognition workflow

3.1 Face Detection

Initially, the process starts with face detection, a computer vision technique focused on identifying and locating human faces within images or video frames. It involves the application of algorithms and models that analyze visual data to determine the presence and precise location of faces, often delineating facial features like eyes, nose, mouth, and the overall face contour. These algorithms can range from traditional methods like Haar cascades to more sophisticated deep learning-based approaches such as MTCNN (Multi-task Cascaded Convolutional Networks) or Single Shot Multibox Detector (SSD). Face detection serves as a crucial initial step in various applications, including facial recognition, surveillance, biometrics, and augmented reality, laying the foundation for subsequent facial analysis and recognition tasks within these domains.

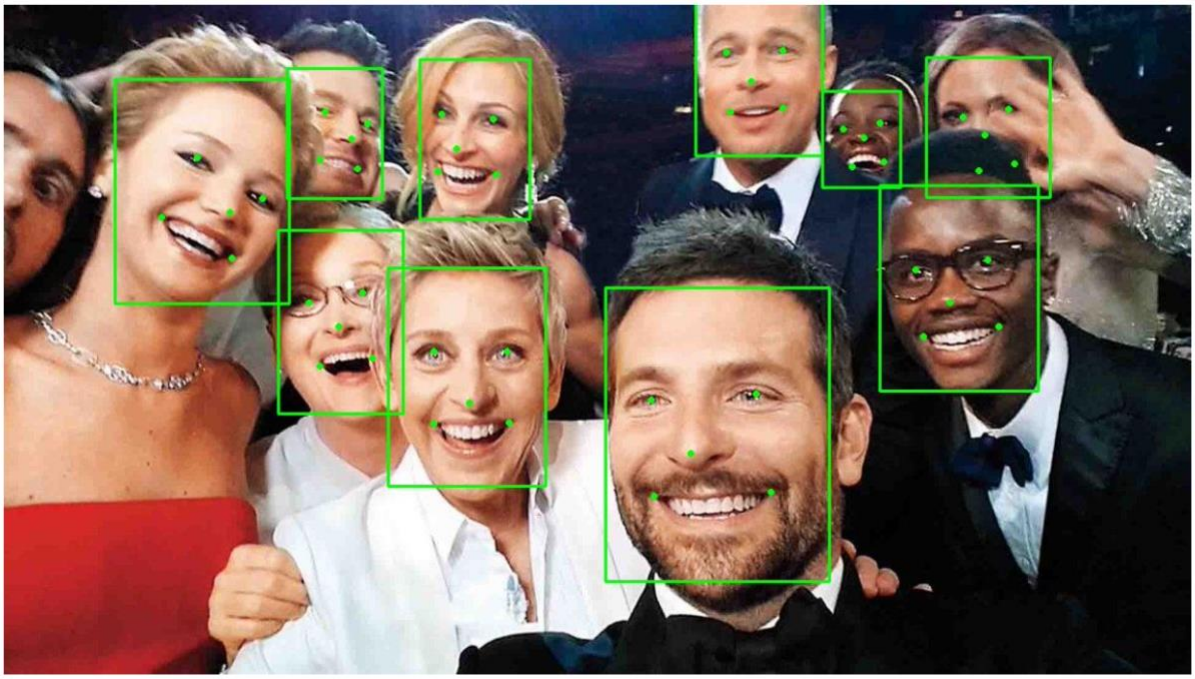

Figure 13: Example of Face Detection on the famous Oscar 2014 selfie.

3.2 Feature Extraction

Feature extraction in the context of face recognition involves the process of capturing and representing unique facial characteristics in a numerical form, often referred to as embeddings or feature vectors. This technique extracts discriminative information from facial images, allowing for meaningful comparisons between faces. Methods like deep learning architectures—such as Convolutional Neural Networks (CNNs) including ResNet, VGG, or Siamese networks—are commonly employed to extract these features. These networks learn to identify and encode facial patterns, landmarks, and textures, creating compact and informative representations that capture essential details while discarding redundant or less informative elements. These extracted features, in the form of embeddings, facilitate accurate comparisons between faces, enabling tasks like face recognition, verification, or similarity assessment, and play a pivotal role in establishing robust and reliable facial analysis systems.

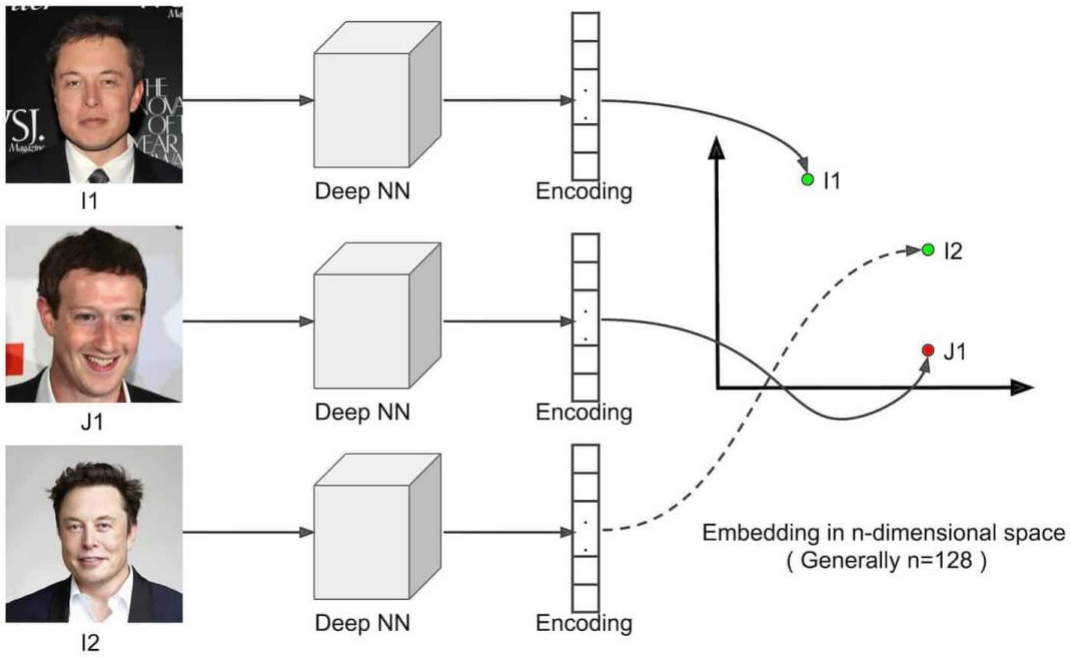

Figure 14: Example of feature extraction

3.3 Recognition

In the final step of face recognition, extracted facial features or embeddings undergo comparison and recognition processes. These representations, often numerical vectors capturing distinctive facial traits, are used to differentiate and match faces within a dataset. Similarity measures like cosine similarity or Euclidean distance are commonly employed to quantify the likeness between embeddings. This comparison enables the system to verify if a queried face aligns with any known identities in a database or to discern similarities between faces. This recognition step involves setting similarity thresholds to determine if the similarity score surpasses a predefined value, signifying a recognized identity. By leveraging these extracted features and employing similarity metrics, the recognition phase enables the system to perform identity verification, classification, or similarity assessments, underpinning various applications like access control, surveillance, personalized services, and social media functionalities.

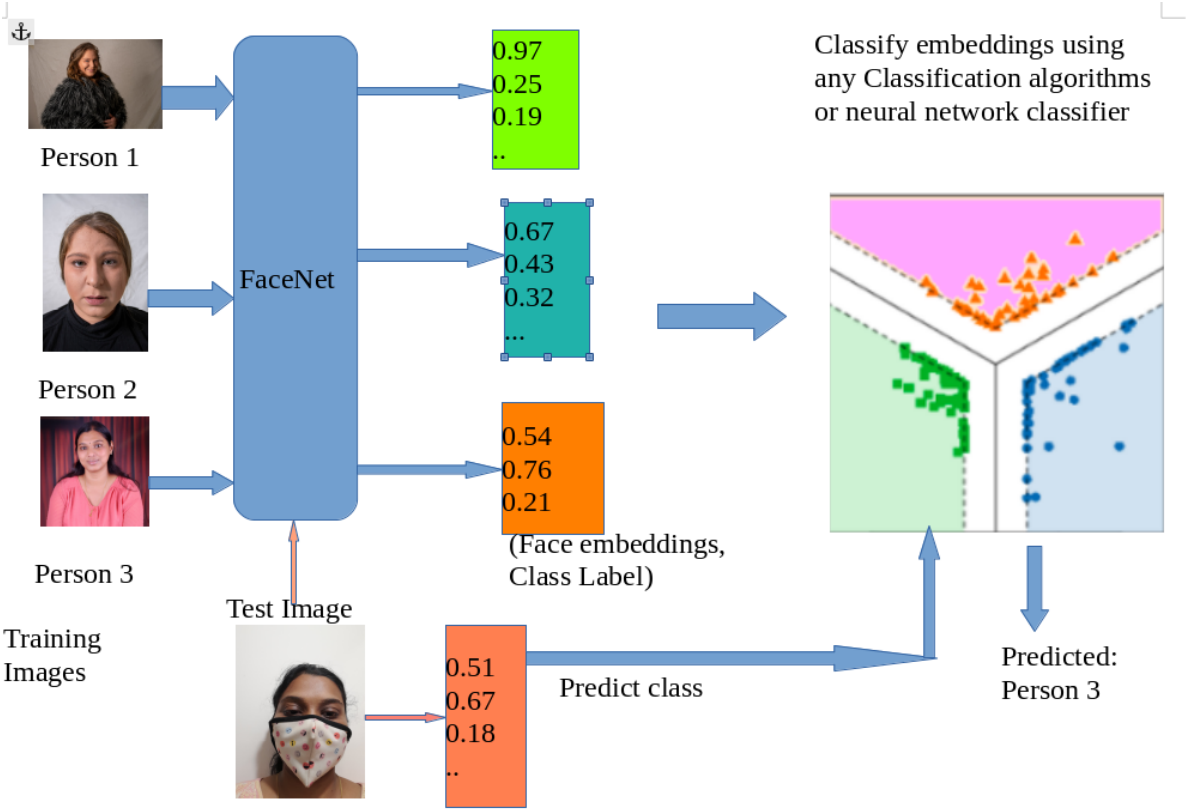

Figure 15: example of recognition based on embeddings.

5. Implementation

4.1 Dataset

Since there wasn't a well-structured dataset available online for the project's purposes, it was decided to create one specifically. The proposed network was trained on a dataset comprising 100 classes, each representing an actor (selected from the top 100 most influential actors according to the IMDB ranking). A high number of classes was chosen to simulate real-world application usage. To ensure a high precision rate, each class contains approximately 100 images.

```
response = google_images_download.googleimagesdownload()

def imbd_function(num_actors):
    num_pages = num_actors // 50
    print(num_pages)
    names_list = []

    for page in range(num_pages):
        my_url =
            'https://www.imdb.com/search/name/?match_all=true&start={}&ref_=rlm'
            .format(page * 50 + 1)
        imdb = urlopen(my_url)

        bsobj = bs.BeautifulSoup(imdb.read(),'html.parser')

        pic = bsobj.findAll('img')

        for img in pic:
            name = img.get('alt')
            names_list.append(name)

    return names_list
```

Figure 16: download actor images (part 1)

The dataset was built using a Python script that dynamically scrapes the webpage https://m.imdb.com/chart/starmeter/. It downloads hundreds of images of the top 100 actors in order of popularity.

```
def download_actor_images(query):
    directory_name = query.replace(" ", "_")

    arguments = {"keywords": query,
                 "format": "jpg",
                 "limit":200,
                 "print_urls":True,
                 "image_directory": "{}".format(directory_name),
                 "size": "medium",
                 "aspect_ratio":"panoramic"}
    try:
        response.download(arguments)

    except:
        print("Image impossible to download")

def download_images(num_images):
    names_list = imbd_function(num_images)
    print(names_list)

    for name in names_list:
        download_actor_images(name)
        print()
```

Figure 17: download actor images (part 2)

The script will create a directory for each actor, applying the appropriate label to each file.

4.2 Neural Network

Although it is possible to use two distinct models: one for the embeddings and the other for the classification task; the following solution will use an end-to-end model which does both the tasks at the same time.
The advantages of using a single model are: easier management and deploy of the model. On the other hand, having two separate model can present a modular solution and can be better fine tune on the specific components.

4.2.1 Training

The training of the neural network has been done using TensorFlow APIs integrated into AWS SageMaker. The actual code for the training in the Jupyter notebook is very concise.

The core logic for the neural network and the training process reside in a different training script (training_script.py). This approach simplifies TensorFlow usage, allowing the utilisation of its standard version without necessitating any modifications.

```
import sagemaker
from sagemaker.tensorflow import TensorFlow

sagemaker_session=sagemaker.Session()
role = sagemaker.get_execution_role()

train_data_location = 's3://{}/train'.format(bucket_name)
output_path = 's3://{}/output'.format(bucket_name)

estimator = TensorFlow(entry_point='training_script.py',
                       role=role,
                       train_instance_count=1,
                       train_instance_type='ml.p2.xlarge',
                       framework_version='2.6.0',
                       py_version='py38',
                       output_path=output_path,
                       hyperparameters={"epochs": 10},
                       sagemaker_session=sagemaker_session)

estimator.fit({'train': train_data_location})
```

Figure 18: sagemaker training

As its possible to see in the previous snippet of code about the training code on SageMaker, there are some parameters which can be defined:

- Train instance count: this parameter dictates the number of instances allocated for training. Increasing this count can accelerate training by distributing the workload across multiple instances concurrently. However, scaling up the number of instances can also escalate costs, especially for larger counts.
- Train instance type: the chosen instance type influences the computational resources available for training. Using a larger instance type can enhance training speed due to increased computing power and memory. On the

other hand, this choice can impact costs, as higher-performing instances have higher charges.

- hyperparameter: changing hyperparameters, such as the number of epochs, allows fine-tuning the training process without modifying the core training script. Adjusting the number of epochs affects the number of iterations through the dataset during training, potentially impacting the model's convergence and performance.

While using an higher count of training instances or opting for larger instance types can accelerate training, it's essential to consider the trade-off between speed and cost. Increasing resources for faster training can significantly elevate expenses, especially for prolonged or resource-intensive training jobs.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--epochs', type=int, default=os.environ.get('SM_OUTPUT_DATA_DIR'))
    parser.add_argument('--model-dir', type=str,  default=10)
    args, _ = parser.parse_known_args()

        class_count = 100

    data_path = '/opt/ml/input/data/train/'
        with np.load(data_path + 'dataset.npz') as data:
                train_faces = data["arr_0"]
                train_labels = data["arr_1"]

        face_images = preprocess_input(np.array(train_faces))

        out_encode = LabelEncoder()
        out_encode.fit(train_labels)
        face_labels = out_encode.transform(train_labels)

        x_train, x_test, y_train, y_test = train_test_split(face_images,
                    face_labels, train_size=0.8, stratify=face_labels, random_state=0)

        base_model = ResNet50(weights='imagenet', include_top=False)
        base_model.trainable = False

        model = Sequential()
        model.add(Resizing(224, 224))
        model.add(base_model)
        model.add(Flatten())
        model.add(Dense(1024, activation='relu'))
        model.add(Dense(class_count, activation='softmax'))

        model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

        hist = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=10, epochs=10)

    model.save('/opt/ml/model/actor-recognizer/2')
```

Figure 19: training script

In the training script I used the ResNet50 architecture trained on the ImageNet dataset. For this purpose, I employed the Transfer Learning technique. If a model is trained on a sufficiently large and general dataset, it effectively serves as a generic model of the visual world, requiring fewer resources and less time in the neural network training process, resulting in an optimal outcome.
The model will incorporate a new classifier (the last layer of the architecture), trained from scratch, allowing the reuse of previously learned feature maps based on a specific dataset. There's no need to retrain the entire model. The base convolutional network already contains generally useful features for image classification. What's required is updating the model's classification layer.
Using ResNet necessitates a large volume of images per class and takes significant time for network training. Utilising tools like AWS SageMaker accelerates the training process by leveraging machines with integrated GPUs.

4.3 Deployment on the edge

Following the training process in AWS SageMaker for a TensorFlow model, the next step involves deploying it into the Greengrass Core device for inference tasks.
With greengrass V2 the deployment of machine learning model is done via components. It’s possible to use a public component and customize the used model. For instance, using the TensorFlow Lite component allows modifications to the public component for custom model integration. To integrate your machine learning models into Greengrass sample runtime components, you'll need to replace the default public model components with customized ones that include your models as part of the system. These components support TensorFlow Lite environment, a more lightweight version of the conventional TensorFlow framework. It's crucial to convert your models to align with this runtime environment.

The following are the custom artifacts used in the JSON recipe to specify the S3 bucket that contains the custom model.

```
"Artifacts": [
    {
      "Uri": "s3://actor-recognizer/model/tflite/TensorFlowLite-actor-recognizer.zip",
      "Digest": "yT1o1yWIcb/iGg2g9V9CdxhXvITu4tSRBzeyuOhoLtM=",
      "Algorithm": "SHA-256",
      "Unarchive": "ZIP",
      "Permission": {
        "Read": "OWNER",
        "Execute": "NONE"
      }
    }
 ]
```

Figure 20: artifacts for deployment in the edge

Other than that, the ModelResourceKey needs to be modified in the inference component to use the custom model.

```
{
    "ImageName": "actor.jpeg",
    "InferenceInterval": "20",
    "PublishResultsOnTopic": "ml/tflite/image-classification",
    "ModelResourceKey": {
      "model": "TensorFlowLite-actor-recognizer"
    }
}
```

Figure 21: inference component changes

4.3 Results

In conclusion, the project's integration of AWS SageMaker for model training, and the deployment through AWS IoT Greengrass, made it easy to bring machine learning capabilities to edge devices. This process simplified the transfer of trained models from the cloud to the edge, reducing complexities significantly.
Using AWS services, it’s possible to mix the advantages of using the cloud for expensive and complex operations (i.e., machine learning or storing large amount of data) and deploying the results of those operations on an edge device with AWS Greengrass reducing the latency in the communication.

The following graphs shows the results obtained during the training of the model. One of the graphs shows the accuracy obtained using dropout which helps the network generalizing on new data, losing some of the accuracy in the training process which holds less importance.

Despite a high number of classes, the test dataset's accuracy is close to an impressive 95% accuracy. This outcome is given by the large number of images used during the training phase.

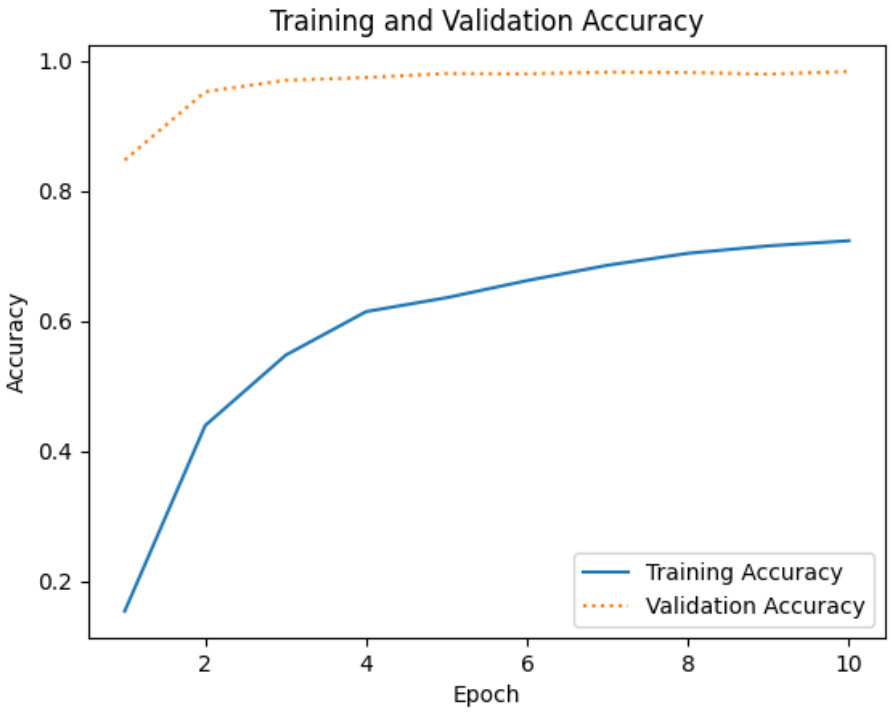

Figure 22: accuracy using dropout technique

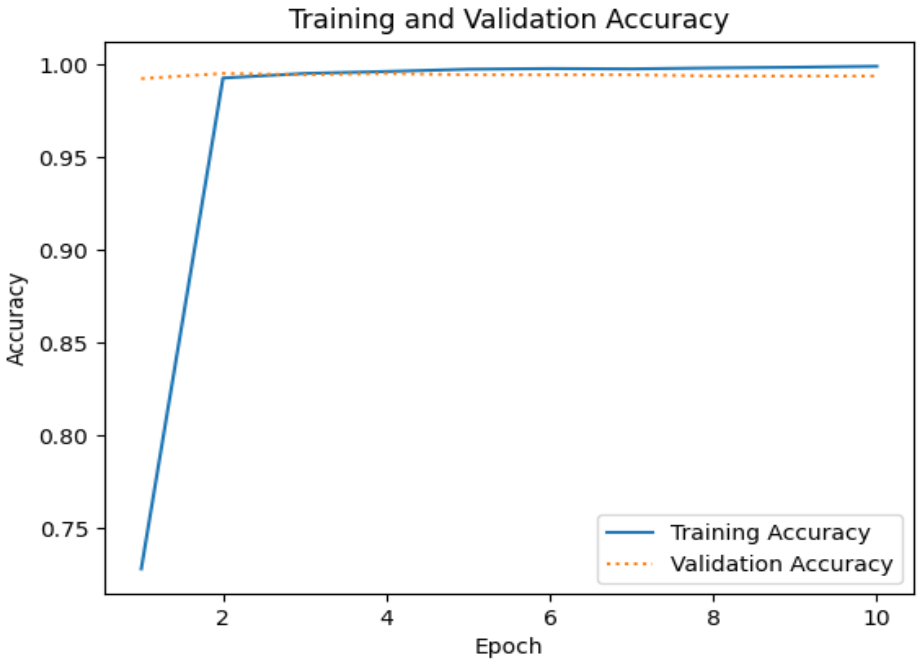

Figure 23: accuracy without dropout